IDA PAPER P-3191

THE COSTS AND BENEFITS OF DOMAIN-ORIENTED SOFTWARE REUSE: EVIDENCE FROM THE STARS DEMONSTRATION PROJECTS

Thomas P. Frazier, *Project Leader*
John W. Bailey

June 1996

Prepared for
Defense Advanced Research Projects Agency

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. H 96-001689

The work was conducted under contract DASW01 94 C 0054 for the Defense Advanced Research Projects Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) for the Defense Advanced Research Projects Agency under a task entitled "Software Environments: Open Architecture Definition and Cost." The objective of the task was to develop and test quantitative measures of the benefits of megaprogramming for three projects designed to demonstrate of the Software Technology for Adaptable, Reliable Systems (STARS) technologies. This paper describes the costs and benefits of the use of such technologies for the three demonstration projects.

This work was reviewed within IDA by Bruce N. Angier and John F. Kramer.

CONTENTS

FIGURES

TABLE

A. INTRODUCTION

The Software Technology for Adaptable, Reliable Systems (STARS) program office asked the Institute for Defense Analyses (IDA) to assess STARS-promoted software development technology. Specifically, IDA was tasked to measure the costs and benefits of domain-oriented tools and processes used in three software projects. These three projects, one each in the Army, Navy, and Air Force, were known as the STARS demonstration projects. The projects began in FY 1993 and ran through FY 1995. The measurement task had two main objectives: (1) to determine in what ways product-line programming occurred on the demonstration projects and (2) to assess the effects of the product-line technologies that were attempted. Now, three years later, we are able to look back and begin to answer questions relative to these objectives.

B. APPROACH TO MEASUREMENT

We designed a goal-directed measurement method to capture such details about project methods and processes as the distribution of labor, the effectiveness of tools and techniques, and the quality of the developed products. This goal-directed approach to measurement design often yielded different metrics than those that might be conventionally used for assessing commercial projects, such as those that reveal overall project efficiency or productivity. Still, we needed to be able to view the results of the demonstration projects at a high enough level to allow assessment in terms that would enable comparisons with conventional developments at the same organization. This meant that productivity analysis was also important.

The extra requirements on project measurement, to assess the effect of product-line technologies as well as to assess the overall project success, led to the specification of far more data collection than was practical, given the duration and size of the projects. For example, using a goal-directed metrics identification approach, initial planners identified 178 possible data items in five categories of product-line assessment: reuse, process, software engineering environment (SEE), planning and estimating, and end product. The Air Force project planners took this as a point of departure and documented over 100 items to help quantify the product produced, the process followed, and the organizational acceptance of the methodology. The Navy project identified 32 quantities in eleven categories to provide a management-oriented view of the project, including certain risk elements, such as requirements volatility, process stability, and SEE response time. For completeness, we also verified that both of these data sets included examples of the four

Software Engineering Institute (SEI) core measures (size, effort, schedule, and quality) [1].

Although the metrics identification processes were effective exercises for focusing the project teams on the value and need for measurement, a comprehensive implementation of such ambitious data collection was not feasible in the time and budget available. Experience with implementing software measurement programs has taught us that data collection must be adopted incrementally and not all at once. Therefore, we examined our data wish-lists and chose those measures that (1) would provide the largest benefit to characterizing the project progress and output and (2) were temporal in nature (which meant we had to record them when they became available or they would be lost). On both the Air Force and Navy projects, we chose to initially collect effort by activity as a way to quantify project investment as well as to help reveal the actual process being followed and the size of the product components as they were delivered. These basic quantities would allow us to compute efficiency, or productivity, as expressed by output divided by input. We planned to add quality measures later, as relevant test and inspection data became available, and to rely on schedule measurement as recorded by project milestones. To this basic set of data, the Air Force added a measure of SEE usage, and the Navy added measures of the intermediate domain products required by the specific development process they were following.

IDA worked closely with the Air Force and Navy on their demonstration projects to conduct its analyses. For various reasons, IDA did not have the same degree of interaction with the Army demonstration project and thus could not conduct an independent assessment of it. Therefore, this paper focuses on the Air Force and Navy demonstration projects.[1]

C. MEASUREMENT IMPLEMENTATION

We had to begin measuring effort as early as possible because we knew that activity data would be difficult to obtain after the fact. Each demonstration project defined categories of effort that were meaningful to the processes being followed. The Air Force team decided to automate effort data collection at SEE log-in time using a pop-up window that would accept a user's declaration of hours spent on various activities. In practice, however, this proved not to be an effective mechanism. One reason was that

---

1 For raw data on the Army project, contact John Willison at U.S. Army Communications-Electronics Command, ATTN: AMSEL-RD-SE-SSL, Fort Monmouth, NJ 07703.

many workers did not use the SEE often enough to remember all their activities between log-ins, and another was that the moment of log-in did not seem to be a natural time to report work done. As a result, much of the early detail about the distribution of activity on the project had to be reconstructed later with management assistance. The Navy project, which implemented a more costly manual reporting process, and which involved a smaller team, was able to collect sufficiently descriptive activity data contemporaneously.

Measurements of the output products were kept to a minimum for the project-wide analyses. Both projects recorded lines of Ada code produced by the domain engineering and application engineering efforts. In addition, the Air Force project also tracked inspection effort and defects discovered and the Navy added measurement of the intermediate RSP (Reuse-driven Software Process) products.

D. THE DEMONSTRATION PROJECTS

The STARS Demonstration projects were selected for their size, schedule, and willingness to use the technologies under evaluation. Certain fundamentals are common to all STARS technologies, but the specific methods employed are not dictated. STARS emphasizes process, reuse, and SEE as the three categories of technology that must be applied to the development and maintenance of software. Specifically, the life cycle must support and enable a product line approach (also referred to as a dual-life cycle or domain-specific approach) to the construction of software. Reuse of domain-specific software assets enables the rapid and low-cost construction of applications or products in the product line. SEE support to the programmer and manager, to automate process where possible and to provide organization and decision support throughout the life cycle, constitutes the third of the STARS-emphasized technology areas.

Given this latitude in the choice of a STARS-compliant suite of software technologies, each of the three demonstration projects adopted tools and approaches that were seen as the most appropriate to its particular project goals. Thus, the three projects, which were each attempting different kinds of development, were conducted in very different manners. For example, the Air Force project emphasized techniques to accelerate the replacement of large amounts of scientific command and control software by exploiting opportunities for horizontal reuse (reusing common functions across different kinds of applications). Alternatively, the Navy project tested methods for developing families of simulator systems and thus attempted a vertical reuse strategy (where a generic system is repeatedly used to develop applications of the same kind). Further, the Air Force project had about a two-year head start on the Navy and Army projects, which meant that

the observation period included far more opportunities to observe benefits than could be observed for the other projects. Conversely, the Navy project attempted the use of a project-wide process and invested considerably in the specification and study of that process.

As a result of these disparate approaches, we conducted only within-project analyses and decided to avoid any between-project comparisons. Therefore, each project was compared with past baselines for similar work by similar organizations and no attempt was made to normalize the definitions for size, effort, or cost between the projects. Because of the different definitions and circumstances within each project, we caution readers against drawing any conclusions about the relative effectiveness of techniques used on different projects. Instead, readers should focus only on improvements relative to the relevant baseline.

The remainder of this section discusses the studied projects. The Air Force demonstration project is discussed first, followed by a brief discussion of the highly productive successor project in the same domain. The Navy demonstration project discussion follows. No successor project was completed in the Navy domain; however, projections based on analysis are given for the expected productivity for such a follow-on project. No discussion of the Army demonstration project is possible for the reasons given earlier.

1. The Air Force Demonstration Project

The Air Force STARS demonstration project was a redevelopment of a portion of the Cheyenne Mountain Upgrade (CMU) at Peterson Air Force Base in Colorado Springs. The CMU program, which began in 1981, is intended to replace and modernize the computer systems at the Cheyenne Mountain Complex. These systems form the nucleus of the worldwide Integrated Tactical Warning and Attack Assessment (ITW/AA) system, which supports the North American Air Defense Command by providing attack information to United States and Canadian leaders. The systems are designed to identify and track potential enemy objects through surveillance, air defense warning, and attack assessment. ITW/AA consists of a worldwide network of ballistic missile, atmospheric, and space warning systems, intelligence centers, associated communications links, and command and control centers. The CMU upgrade program is intended to replace about 12 million physical lines of mainly FORTRAN code. Part of the upgrade is SPADOC 4, a 2-million-line component that performs space surveillance (Because the baseline FORTRAN

system was measured in physical lines of code, the Ada software in this study was also measured in this manner.)

The Mobile Space project, which is the name of the Air Force demonstration project, redeveloped about 50% of the total SPADOC 4 functionality using domain-based development. The complex core functionality, which involves tracking and assessing space objects, was included in the effort. The Mobile Space operator interface was improved relative to SPADOC 4, however the Mobile Space project did not implement certain other aspects of SPADOC 4, such as its security layer, special communications processor handling, and certain event-handling capabilities. The redevelopment was targeted for installation in mobile vans to replicate the CMU software function, hence the name "Mobile Space."

a. Process

As previously mentioned, the Air Force project was already underway when it was funded by STARS to be one of the three demonstration projects. During the two years before the STARS observation period, the project invested in horizontal reuse assets such as code generators, graphical display and interface builders, and database tools. It also developed a reusable architecture with which the current project and future pieces of CMU function could be built. Early in the demonstration period, the Air Force successfully pilot tested its reusable architecture and tools. Therefore, the bulk of the demonstration period required only limited additional domain-specific investment.

Although no rigorous overarching process was defined or followed to conduct the domain and application engineering efforts, a particularly successful implementation of the Cleanroom approach to software development arose from efforts to improve the quality and speed of code production.[2] This code development process was highly specified and it involved tracking each software component through twelve inspectable states, from early specification to certification test. The development manager cites this process, in addition to the reuse tools and architecture, as one of the main reasons that low-defect software could be developed quickly.

b. Cost Summary

In February 1995, an interim briefing on the progress of the project was given by IDA to the Advanced Research Projects Agency Joint Advisory Committee. This briefing,

---

2 See Reference [2] for an explanation of the Cleanroom approach.

"Benefits of Megaprogramming: Preliminary Results from the Air Force Demonstration Project," concluded that the baseline cost of new space application software is about $130 per line, whereas the projected finished cost per tested line of the Mobile Space application was less than half of that cost. This cost included the expenditure by the Air Force of $9.6 million before the beginning of the demonstration period to develop and pilot test application infrastructure code and application generators before developing the main application code.

Now that the Mobile Space project is completed, we can report that it consists of 487,649 physical lines of Ada code, including generated code (109,877 lines), reused code (30,165 lines), and hand-written Ada (347,607 lines). Code-generator and infrastructure tools consisting of an additional 92,000 lines of code were also developed under the same budget. The entire effort investment at the organization from before the demonstration project began (when domain assets in the form of generators and reusable infrastructure were being developed) up to the end of the demonstration project was just under $20.2 million. Of this, $6.7 million was spent for the development of the infrastructure and code generators (92,000 lines), $2.9 million was spent to pilot test the reusable infrastructure, and another $4.8 million was spent for fixed costs and other non-project effort to support the organization. This means that $5.8 million was directly spent to develop the 488,000 delivered lines of code. (These figures are for labor only, and do not include the cost of equipment, space, or utilities. However, they do include overheads for labor benefits such as insurance, training, and retirement.)

Using the $5.8 million figure for total cost, it might appear that the net cost for the software is around $12 per physical line. To be thorough, however, we also included the organizational support costs and the cost of the infrastructure tools and their testing in order to account for the total effort required to produce the 488,000-line project. This raised the total cost to $20.2 million and the cost per line to $41. However, neither developmental test and evaluation (DT&E) nor operational test and evaluation (OT&E) was conducted. Earlier software products in this organization have expended 40% of their total development cost in OT&E and 20% in DT&E. Even though the defect discovery rates were low, we have no reliable data from earlier products that would allow us to conclude that testing effort would be lower than these earlier reported figures (see the next subsection, Quality Summary).

We chose to correct for the full measure of presumably missing effort by assuming that DT&E and OT&E both remain to be conducted. This means that the $5.8 million spent directly on this project represents only 40% of the total that might be spent by the

end of OT&E. This brings the total corrected cost to $28.9 million, which is the $20.2 million plus an additional $8.7 million (the other 60%) for testing. This increases the final cost per line to about $59 per line. In some other reports about this project, the entire measure of Ada produced on the project, including the infrastructure tools (about 92,000 fully tested lines), is reported as project size. Counting in this way means that 580,000 physical lines of Ada were produced, resulting in a total adjusted cost of $50 per produced line. We did not include these lines as delivered because they amount to tools rather than project code. We therefore initially reported a corrected figure of about $59 per line of Ada.

To be fair, we must note that the Ada code was not the only software needed to accomplish the Mobile Space mission. In addition, 29,308 lines of Structured Query Language (SQL) were written by hand and about 168,000 lines of query builder tool (QBT) and display builder tool (DBT) input files were generated from high-level programming. All of these additional lines of special-purpose software serve to off-load the requirements on the general-purpose Ada programs for a net gain in programming efficiency. However, this means that the 488,000-line figure used for program size is too low to represent the functional size of the software in conventional terms. We decided to add the SQL size since it was hand-developed and represents at least as many lines of Ada in terms of the function embodied by it. However, we did not have so strong a justification for adding the QBT and DBT lines since they are primarily generated and act more like data than lines of software. Adding the SQL lines lowers the $59 cost to $56 per line of software.

For the Ada delivered on Mobile Space, the ratio of physical lines to non-blank, non-comment lines is just under 2:1, while the ratio of physical lines to Ada semicolons is slightly more than 3:1. We did not apply any correction to compare the size of an Ada program in physical lines with its FORTRAN counterpart. Other work at IDA has shown that, although small Ada programs are more verbose than functionally equivalent FORTRAN programs, several factors result in an economy of scale which benefits Ada in the case of larger programs [3].[3]

To demonstrate the economy of Ada, the Air Force development team shows numerous examples of how reuse and efficient design led to programs that are more succinct than their FORTRAN counterparts. Since Mobile Space reimplements roughly

---

[3] The economy of scale realized with Ada has also been observed at NASA/Goddard's Flight Dynamics Division, which has extensive experience with Ada.

half of SPADOC 4, which is about 2 million physical lines of FORTRAN, we remain comfortable that the Ada sizes reported here are not greater than the sizes of equivalent FORTRAN programs. In fact, they are probably smaller by perhaps as much as 50% using the rough figures for SPADOC 4 size and function. So, if the sizes of the Ada programs are adjusted upward as a way of estimating size as measured by FORTRAN lines of code, then the cost per line would actually be lower than the $56 per line reported here.

Even the upper bound cost of $59 per line (which does not give credit for the SQL lines) compares favorably with the baseline cost of $130 per tested physical line for conventionally developed systems at this site, which is a figure we derived from knowing the cost of developing SPADOC 4 software. (Unsubstantiated cost estimates from CMU contractors for fully documented and tested systems were stated as high as $400 per FORTRAN line. This figure may be based on executable FORTRAN statements rather than on all physical lines, however, which would make it consistent with our estimate of $130 per physical line.) We confirmed our baseline cost figure using the Government Accounting Office estimates of the total CMU cost. We also used the COCOMO cost model. Because we believe the correction for testing we applied to the Mobile Space development is probably too conservative (i.e., it overcorrects for the unperformed testing), and since much of the cost would not have to be repeated for another system in the product line, we feel the actual incremental cost of developing related systems will be even lower than these figures. Fortunately, we were able to obtain data from a follow-on development by the same organization using many of the same tools and techniques, and these data confirmed our suspicions. Those results are reported in the section following this discussion of the Air Force demonstration project.

c. Quality Summary

During our analysis, we made conservative assumptions about the quality of the code in order to compare the results with the costs of conventional software development. Our best information is that the baseline FORTRAN system suffers from about three defects per thousand lines of code. Testing of the Mobile Space software revealed 398 errors. Because walk-throughs had been conducted as part of the Cleanroom development process used, and 1,700 major and minor defects had already been uncovered and corrected, we are comfortable that the error rate reported during testing is reasonable. Relative to the walk-through defects discovered, the rate is neither too low, which would suggest that insufficient testing had been performed, nor too high, suggesting that the walk-throughs let too many errors through and that further testing would reveal even

more errors. A conservative rule of thumb is to assume that testing uncovers about half of the defects remaining in a piece of software (although many of the remaining defects may never be noticed).[4]

If we assume that about 400 errors remain in the 517,000 delivered lines (including the SQL), then we would expect that less than one defect remains per thousand lines of delivered code (0.8 defects per thousand), and many of those might never be encountered in operational use. This exceeds the reported quality of the baseline system by nearly a factor of four. Further, the Ada developers identified certain previously unknown errors in the existing, operational FORTRAN system that were implemented correctly in the Mobile Space redevelopment. We conclude, therefore, that the quality of the Mobile Space implementation, as measured by defect density, is several times better than the baseline system. However, to prove the validity of the productivity improvements, we need only show that the quality is *no worse* than that of the baseline.

d. Problems and Open Issues

Several fundamental measurement problems arose regarding the quantification of system size, and several aspects of these problems were never completely resolved. In order to generate productivity and cost assessments, we made assumptions to address these problems. Whenever we had any doubt, we made one-sided assumptions in that we always chose them to make the demonstration project look either more costly or less productive. In spite of these assumptions, the demonstration project still appeared to be significantly more productive, or less costly, than previous comparable development efforts.

We found that the line between reuse and use had been blurred when we attempted to finalize the size of the reusable command and control architectural infrastructure (CCAI) code. In previous analyses and presentations, we included the size of these components at the time of the pilot project, which was 64,440 lines of Ada code. This included the universal network architecture services (UNAS), the reusable human-machine interface (RHMI), and the reusable integrated command center (RICC) tools. At the completion of the project, we learned that the tools no longer resided in the project libraries but were instead made available to the project as packaged, supported, and licensed products in object code form with their source code being managed by TRW at

---

4 Based on unpublished testing and debugging research conducted by one of the authors at General Electric Company in 1980.

no additional cost to the project beyond the license and support fees. This presented a curious dilemma since some of the development of those tools was paid for by the Air Force before the pilot project, and those costs have been included in our analysis. We could continue to credit the project with 64,440 lines since that was the count the last time the tools were considered project assets, or we could ask TRW to estimate their current size (which tools we did—the three tools now consist of about 92,000 lines). Alternatively, we could declare that these products have been promoted from examples of project-managed reuse to commercial off-the-shelf (COTS) products and as such we should no longer include their source lines as part of the developed project, just as we would not count the source lines of an operating system or other external product.

This illustrates what was probably the most significant difficulty we encountered when trying to compare the product-line demonstration development with conventionally produced existing software. Since software development technology has been improving over time, it is important to know the context for any baseline assessments. If we are simply comparing the size of the demonstration project with the existing conventionally developed (i.e., 1980s technology) SPADOC 4 software, then we need to count this reused infrastructure software as part of a functional size measure. However, if we are assessing the productivity of product-line programming, where methods and tools reduce the need to write unique lines of code, then we can ignore these infrastructure lines as long as we understand that each developed line of code delivers more function than lines developed in settings where nearly all function is hand-coded (with the exception of standard input/output, math, and database routines).

Another issue with respect to system size is whether to count the non-Ada lines of screen management data and query building data (DBT and QBT), which are generated and are read in by the system as it executes. One argument says that these lines are produced as a result of programmer effort in the form of pointing and clicking a mouse and typing in a few lines directly. TRW says the effort to run the generators to obtain these lines is about 10% of the effort it would take to write the lines without the tools. So, we would like the value of the delivered product to reflect this effort by counting the data lines as generated lines of code. Another argument says that screen builders and query builders are commonplace (familiar examples would be Microsoft's Excel and Access); therefore, counting the lines of data produced by those tools artificially inflates the apparent project size. We did not include any of these non-Ada lines in the representations of project size used in this report.

Figures 1 and 2 are adaptations of diagrams that the development manager of Mobile Space uses to illustrate the difference between line-oriented programming and tool-based program development. The area above the horizontal line in each case shows what is submitted to the compiler. In the conventional situation, shown in Figure 1, all source code to be compiled is written by hand. After compilation, the object code is linked with libraries to support input/output, math, and database manipulation. In an environment with productivity-enhancing tools such as screen builders and database query builders, not only are many of the compiled lines generated, but additional reusable libraries exist as part of the operating system.

If one ignores the generators, the generated lines, and the additional linkable libraries when assessing the size of the delivered product, then productivity comparisons using conventional line-of-code counts (from situations as in Figure 1) lose their meaning. We counted the generated Ada lines but had no reliable way to fairly assess the "size" contribution from other tools and object code libraries. One sample of display software from Mobile Space showed a compression of more than 3:1 when comparing the number of lines required to do the same function in a more conventionally produced program. Although we could not assume this much compression applied to the entire Mobile Space product, this did seem to demonstrate that using counts of source code lines to measure delivered function on a tool-supported reuse-oriented project underestimates the true size of the software produced. Had we been more successful determining a calibration factor to compensate for this compression, the corrected size of the delivered product, after normalization for comparison with the baseline FORTRAN system, would have been higher.

At one point we were hoping to be able to measure system size independently of the number of lines of code in the implementation. One way to do that is by counts of the number of requirements satisfied, as long as that proves to be a stable and sufficiently well-defined notion. However, one problem with tracking progress in this way is that the specifications for the builds were allowed to evolve, usually by reducing the number of requirements to be satisfied, as development proceeded. Other methods of measuring system size, which may also be applicable to baseline projects, include counts of screens, queries, and sessions. Several of these methods were considered but not seriously pursued. The one success we achieved was a rough calibration of the number of lines of conventionally developed code (Figure 1) required to replace the function of a single line of hand-written code in a tool-supported reuse-oriented environment (Figure 2). This

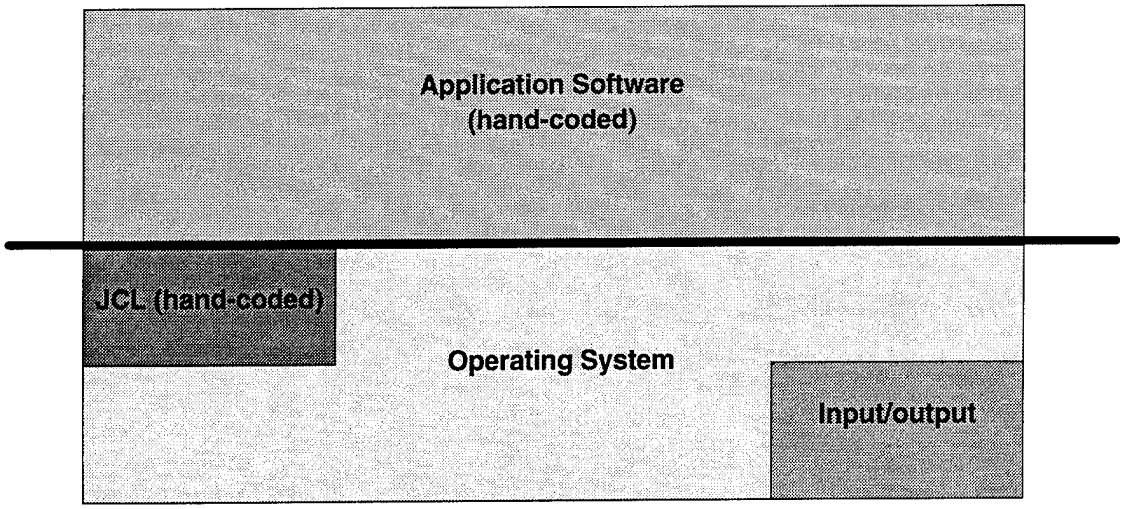

Figure 1. Software Architectures of the Past

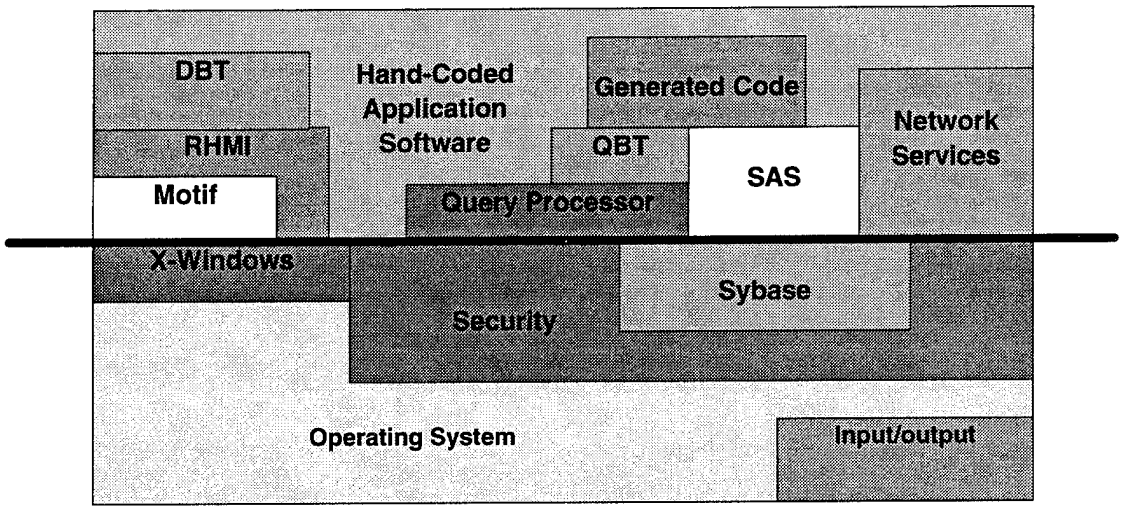

Figure 2. Mobile Space Software Architecture

calibration was based on a well-known portion of the display software and showed a compression ratio of over 3:1 (26,000 lines in the older system was implemented with only 7,700 lines using the reusable architecture). If such code reductions occurred frequently across the 488,000-line project, we would have to correct this traditional measurement of software value by multiplying the size of those portions of the software by three. This would, of course, reduce the cost per line accordingly.

The collection of effort data also encountered its share of difficulties. When the original plan to track effort by activity and to automate the data collection using Amadeus proved not feasible, we did not have a satisfactory backup plan. Instead, we began to rely on management reports for effort and reconstructed the activities being performed by knowing the primary occupation of each individual. In most cases this was an adequate approximation of the data we originally had hoped to get. Because most individuals were dedicated to either domain engineering (DE) or application engineering (AE), we felt we were able to adequately approximate the overall ratio of domain to application activity. We had originally expected a higher proportion of DE during the demonstration phase, although in retrospect we realized that there is no reason that DE and AE have to be performed simultaneously. What happened on this project is that an early DE effort culminated in a pilot demonstration of the capabilities, which was then followed by a primarily AE effort.

e. Summary of Air Force Project Results

The picture that has emerged from this effort is that up-front domain and infrastructure investment can, in fact, be leveraged into increased production efficiency. The project recouped the process and domain investments faster than we expected. Whereas we had expected to be able to report only a positive trend by the conclusion of the demonstration phase, the project showed a high incremental productivity for its most recent output, and passed the break-even point by recovering the entire two-year investment in domain assets just over one additional year.

2. The Successor Project in the Air Force Product Line

For nearly the past year, beginning before the end of the Air Force demonstration project and continuing as of the time of this report, a successor project in the space command and control domain has been reusing the same infrastructure tools and code that was used by the Mobile Space project. To date, the project, known as ATAMS (Automated Tracking and Monitoring System), has delivered 62,058 physical lines of

hand-coded Ada. This executes in conjunction with 8,798 generated and 8,236 reused lines of Ada, and with 11,543 lines of hand-coded SQL, in an architecture similar to that of the Mobile Space software. (The QBT and DBT files for ATAMS, which amount to over 26,000 lines, were not included in the size computation.) The cost of this 90,635-line development has been about $1.1 million, so far. This figure includes $742,000 charged by the various contractors involved and another $325,000 for the cost of government oversight and organizational support through the first quarter of 1996 (using $100,000 as the cost for a burdened government staff-year).

The first release of this software is now operational. DT&E was finished by early March and OT&E is 75% complete. If we use the 40% figure for OT&E, this means that all but 10% of the fully tested software cost is accounted for by the $1,067,000 spent to date. The projected total cost of the current release of ATAMS is therefore $1,186,000, meaning the software cost is just $13 per delivered, tested line. A size factor which was not taken into consideration is that the design required less code than the most closely related prior system by eliminating various redundancies. This means that the function delivered should be thought of as actually larger than 90,635 physical lines as measured by the size of more conventionally designed systems of equivalent function. If, for example, we were able to report the same compression factor for the portions of the project that were similar to those compared in our earlier example, we could claim that the effective size of those parts of the system would be up to three times larger. This means the cost per normalized line of function would almost certainly be less than $13, which already represents a dramatic improvement from the $130 per line accepted as the cost of delivering software at this organization.

The quality of the ATAMS project is also worth reporting. After the internal walk-throughs, which found about 300 defects, the software went through four stages of testing: internal development test, independent certification test, DT&E, and finally OT&E. The number of defects found at each stage has been steadily decreasing. Whereas the developers found 112 errors, the certification test found 91 problems (including several operational anomalies that were not software errors). DT&E subsequently discovered 18 problems and OT&E has discovered only 10 problems so far. Applying the same rule of thumb as before, we might expect to find about 10 more errors in the current system. This would put the total error density of the system at turnover at 38 errors (18 plus 10, plus an undiscovered 10) in 91,000 lines of code, or 0.4 per thousand lines. The predicted error density for the fully tested, operational system would be only 0.1 errors per thousand lines (10 errors in 91,000 lines).

It is fortunate that the Air Force project had the two-year head start in developing domain assets and its architectural infrastructure because it afforded us the opportunity to observe a longer-term view of benefits than would have otherwise been possible. The break-even point, where the conventional cost for the amount of software delivered equals the total project cost including all investments, was observed at just over three years of effort. However, the cost of code production from that point forward was not only cheaper by as much as ten fold when compared with the baseline, but the quality as measured by delivered defect density improved by more than an order of magnitude.

3. The Navy Demonstration Project

The Navy's STARS demonstration project involved a portion of the software for the T-34 flight instruments trainer, which was being redeveloped as part of a service life extension program for T-34 simulators. A previous study had produced a high-level architecture for simulator software that divides any simulator system into distinct, well-defined subsystems, such as flight dynamics, propulsion, navigation, and so on.[5] The Navy demonstration project developed a subset of these architectural components for the T-34 project using a domain-centric process published by the Software Productivity Consortium (SPC) in Virginia (see below).

Originally, the project team planned to reimplement several of the architectural subsystems and completed a substantial amount of the analysis required for their development. However, time and budget forced the effort to be rescoped and the implementation effort was limited to one fairly complex subsystem, known as flight dynamics. The flight dynamics subsystem is thought to replace about 50,000 physical lines of commented FORTRAN software in a conventionally produced system, although comparable existing systems do not use the same architecture, so an equivalent size assessment for this function is difficult.

The full Ada implementation of the flight dynamics subsystem was completed during the demonstration phase and was successfully demonstrated in concert with the other subsystems, which were conventionally upgraded under a separate effort by modifying the original FORTRAN.

---

5 The architecture is explained in Reference [4].

a. Process

The SPC-developed process adopted by the Navy demonstration project—originally known as Synthesis but now referred to simply as RSP for Reuse-driven Software Process—was being used for the first time on a project of this size and importance. There was some overhead spent on learning about and refining RSP, and certain problems with the process had to be worked out with consultation from the SPC. Because there was no prior data to help estimate the amount of time and effort that would be needed to develop the domain assets using RSP, the project twice reassessed its ability to meet its targets for project completion. Initially, all the simulator subsystem domains were analyzed in order to understand the reuse potential of the different subsystems. Instead of continuing to try to develop the entire simulator system, a subset of the domains was selected for the next steps of the RSP process. After this, the scope was again narrowed so that only a single domain of moderately high complexity, the flight dynamics subsystem, was selected for full implementation.

b. Cost Summary

The Navy project expended about 32,600 hours mainly in domain-engineering activities developing a set of domain assets that were instantiated by application engineering into a 54,000 physical-line application for an additional 640 engineering hours. (The subsystem was 25,000 non-blank, non-comment lines of Ada and 11,000 terminal semicolons.) Correcting the development portion of the total effort for the incomplete testing brings the total effort figure for tested software to 37,742 hours. This total compares with an expected effort of about 15,000 hours for a one-of-a-kind project of this size at this organization. Of the reported 32,600 hours, however, 7,000 hours were spent analyzing other subsystems that were never implemented and 3,600 hours were spent on a pilot study. Also, 7,400 hours of the total were spent refining and extending the RSP process used. Of these, we would assume that the first two categories would not be repeated for subsequent applications, and that only a portion of the process expense would continue to be required.

What is unknown is whether the domain assets will be reusable without significant and costly enhancements. The engineers and analysts have studied the requirements for both the T-44 and the T-45 flight dynamics subsystem and feel that the assets used for flight dynamics for the T-34 embody at least 85% of the function required for each of these two additional trainers in the domain. If we accept this, and assume that the additional 15% functionality can be added for the same cost per line as the initial 85% and

that the instantiation costs (application engineering) are comparable to those for the T-34, then the initial investment would be more than recouped after one additional project, when compared with the expected costs of these kinds of projects. Unfortunately, this project did not have the two-year head start that the Air Force project had, so we have no data on successor projects in the domain, which might have shown the long-term viability of the RSP approach.

c. Quality Summary

As was the case with the Air Force project, the Navy project's use of an existing system as a reference for the development of the experimental software allowed a functional comparison between the new and old systems. Also comparable to the Air Force project was the fact that anomalies were discovered in the existing system that were fixed in the newly developed software, resulting in a demonstrably more correct implementation. Most notably was the presence of a discontinuity in the existing FORTRAN algorithm, which caused the simulated aircraft to behave incorrectly during certain roll maneuvers. In addition to this algorithmic bug, the density of the defects discovered so far in the experimental software is lower than that for the conventionally developed system, according to the testers, increasing our confidence in the validity of the productivity figures.

d. Problems and Open Issues

One of the most difficult problems in assessing the cost effectiveness of the domain-centric, dual-life-cycle RSP approach to software development demonstrated by the Navy project was the quantification of a conventional baseline software development cost. None of the contracts for existing simulator systems separated the cost of the software from the rest of the system. Our first attempts to isolate the cost of software per line of FORTRAN by using conventional cost models were arguably too high. Instead we adopted an approach that estimated the baseline software cost by the number of personnel assigned to software activities. Also, because of the complexity introduced by the different pay scales between government, contractor, and consultant labor, we changed the comparison to be in terms of hours instead of dollars per line of code. Since previous IDA studies suggested that the cost and functionality of a line of FORTRAN and a line of Ada are roughly comparable, we did not adjust for language [3].

Another issue with the cost/benefit analysis was the difficulty of separately estimating the overhead cost of using the process for the first time and the cost of the

unneeded requirements analysis. The high-level summary of the cost analysis given above makes no attempt to remove these effects, because we again followed the rule that worst-case assumptions would be made whenever we were in doubt. When the analysis is done in a less conservative manner by removing the cost of analyzing the undeveloped domains, the estimated cost of refining the process, and the cost of the pilot project, the total effort required drops by one-third relative to the actual data collected on the first project. This would give a more realistic upper bound for the cost to develop assets for additional domains in this environment. The 15% figure we estimated as the relative amount of work required to develop additional applications from an existing set of domain assets was derived from the experiences of the demonstration project. The analysis of the T-34, T-44, and T-45 trainers showed about an 85% overlap in function between each successive pair of applications, which meant that about 15% of the work for each of the two remaining simulators would be unprecedented. We do not necessarily expect an increase in size, although the new function might augment as well as replace the existing software.

e. Summary of Navy Project Results

The developers of the Navy demonstration project software followed a rigorous process for domain-based software engineering. Since this was the first serious use of the process outside of the research institution that designed and developed it, there were several time-consuming issues with its adoption and use. The Navy project is therefore credited with considerable refinements and improvements to the process. However, the cost of working with the SPC on these refinements, as well as the effect of the discontinuities in Navy funding for the trainer simulator project, noticeably increased the cost of conducting the project.

Even if no correction is made for these extra expenses, the break-even point for RSP as implemented on this demonstration project will be just after the second application that uses the domain assets. These results are close to, or even better than those reported by other studies of the cost effectiveness of domain-oriented development [5]. Further, if the expenses are adjusted to show only the relevant portions of the cost to more accurately represent the expected cost for a second use of RSP on a new domain within the same organization, the cost of domain development and the first application is only about 50% higher than the cost of a conventional project. Then, because the cost of further reuse of those domain assets for another application in the domain is low (any additional DE work required plus the AE effort to generate each additional application), a savings would be realized as soon as the resulting domain assets were used a second time.

In the projections used here, the incremental effort for a second application would be 15% of the effort to produce the initial system, or about 23% of a conventional development (since the initial system cost 150% of the cost of a conventional development), plus another 2% for the AE effort. This puts the incremental cost for additional systems at about 25% of the cost of a one-of-a-kind system. So, whereas the first system in a domain developed using RSP appears to cost half again as much to produce as a conventionally developed equivalent system, each additional system can be expected to cost only one-fourth as much.

E. RESULTS SUMMARY

This study looked at two of the three STARS demonstration projects. These projects used different approaches to implementing the STARS technologies' emphasis on process and reuse with support from a software engineering environment (SEE). The Air Force approach emphasized up-front investment in general infrastructure tools that accelerated development of the kinds of command and control applications common to the demonstration project domain. The Navy approach involved following a well-defined process for simultaneously analyzing and building generic software for multiple sets of requirements so that each resulting application could be developed inexpensively. The Air Force project enjoyed a two-year head start by investing in reusable infrastructure tools well before the STARS demonstration funding became available, and thus was able to report some of the longer-term benefits from these investments by developing a second project in the same domain. Although the Navy project was shorter lived than the Air Force project and was therefore able to complete only the first of three analyzed flight dynamics subsystems, analysis shows that most of the work for the second and third applications is already complete. The Navy project also invested in the specification of the second generation of the RSP approach for multi-point applications development to ensure that follow-on efforts could leverage available cost savings.

For its software technology demonstration, including the period before the STARS funding began, the Air Force spent $9.6 million developing and pilot testing application infrastructure code and application generators before the main application code was written. However, enough reuse of this investment was achieved that even the most conservative estimates of product value show that 488,000 lines of Ada and 29,000 lines of SQL, which would have an untested value using the baseline productivity of $29 million, were delivered for only $10.6 million beyond the one-time cost of $9.6 million spent for infrastructure. Even more promising is the subsequent fully tested delivery of a

second 91,000-line product in the domain for slightly more than $1 million, a 90% savings from the established baseline.

On the smaller Navy project, 37,100 hours (including the correction for testing) were required to build a set of domain assets for simulator flight dynamics. These assets were instantiated in the first application in 642 hours and are expected to require fewer than 5,000 additional hours to satisfy the flight dynamics needs of a second application. By conventional methods, these 54,000 physical line subapplications would have taken an estimated 19,000 hours each, indicating that even in this relatively inefficient first demonstration of a dual-life-cycle process, the cost of the initial domain effort will be returned after the second application is generated. The fact that the incremental effort of a new application is projected to be only one-fourth of the conventional effort (5,000 hours versus 19,000 hours) is a further encouraging result. This means that after the two-project break-even point, the cost of any additional applications is reduced by nearly 75%.

In the analysis of both projects, conservative assumptions were made about the quality of the code and the time required to fully test and deliver the products in order to compare the results with the full life-cycle costs of conventional software development. In all cases, the demonstration data showed the product quality to be at least as good as comparable conventionally developed software, although the higher-quality code was not used to justify reduced cost estimates to fully test the software. Nevertheless, the current defect data suggest that we will continue to see improvements in terms of testing time and defect density as compared with traditional software deliveries.

Tables 1 and 2 summarize some of the cost and size data for the two projects. In the tables, the top rows show the estimated normalized size of the function delivered (as would be measured by conventionally developed FORTRAN lines of code) followed by the estimated cost (or effort) and schedule to develop this function using the baseline approach. Although data are normalized within each project, the different definitions of size and expenses included in each baseline made it impossible to normalize the data between the projects.

Table 1. Air Force: Infrastructure Tool Investment Study

	Size	Cost	Schedule	Productivity	Relative Cost
Conventional Baseline	800,000	$104 million	107 months	$130 per LOC	100%
STARS Technology	516,957	$29 million	60 months	$56 per LOC	43%
Second Application in Domain	90,635	$1.2 million	12 months	$13 per LOC	10%

Table 2. Navy: Multi-Point Solution Study

	Size	Effort	Schedule	Productivity	Relative Cost
Conventional Baseline	54,000	18,563 hours	22 months	0.34 hours/LOC	100%
STARS Technology	54,000	37,742 hours	26 months	0.7 hours/LOC	203%
Second Application in Domain (estimated)	54,000	5,000 hours	12 months	0.09 hours/LOC	27%

The next rows in each table show the actual sizes, costs, and schedules observed up to the end of the STARS demonstration period, which represents two additional years of work for the Air Force because of its early start. In order to make them comparable to the baseline costs, these figures are corrected for any unperformed testing, which increases the reported costs. The bottom rows in each table show the values for a second application in the domain. The Air Force values are actual, but the Navy values were predicted assuming the project were to continue for an additional year.

The picture that is emerging from these projects is that either of two different styles of domain investment can be leveraged into increased production efficiency. The Air Force project demonstrated that infrastructure investments can be leveraged across multiple applications in a broadly defined domain. The Navy project demonstrated the large reduction in application engineering costs that could be achieved by building generic software for multiple applications within a narrow domain. Both projects indicated that a recovery of domain investments can be expected after about three years of effort. In the Air Force study, domain investments began two years before the start of the STARS demonstration phase, and these costs were recovered after an additional year. In the Navy study, work was interrupted for the first year of the demonstration phase so only two productive years were observed. However, even if we assume only half the rate of expenditure observed during the demonstration phase, the second subsystem in the domain could easily be completed in an additional year, thus recovering of the first two years of investment. That this three-year break-even point was observed in both cases, even though the Air Force project was considerably larger than the Navy project, suggests that the size of the investment may be less important than the duration. In both cases, further development in the same domains can be accomplished at substantially reduced costs, with savings of from 75% to 90% when compared with the cost of conventional software development at the same sites.

REFERENCES

REFERENCES

[1] Software Engineering Institute. "Core Measures for Software Development." Carnegie-Mellon University, 1992.

[2] Mills, H. D., M. Dyer, and R. C. Linger. "Cleanroom Software Engineering." *IEEE Software*, September 1987, pp. 19–25.

[3] Frazier, Thomas P., John W. Bailey, and Melissa L. Young. "Comparing Ada and FORTRAN Lines of Code: Some Experimental Results." Institute for Defense Analyses, Paper P-2899, November 1993.

[4] Crispen, Robert G., Brett W. Freemon, K. C. King, and William V. Tucker. "A Domain Architecture for Reuse in Training Systems" in *Proceedings of the 15th Interservice/Industry Training Systems and Education Conference*, pp. 659–668.

[5] Bailey, J. "A Component Factory for Software Source Code Re-Engineering and Reuse." Doctoral dissertation (UMI 9234514), University of Maryland, College Park, May 1992.

ABBREVIATIONS

ABBREVIATIONS

AE	application engineering
ATAMS	Automated Tracking and Monitoring System
CCAI	command and control architectural infrastructure
CMU	Cheyenne Mountain Upgrade
COTS	commercial off-the-shelf
DARPA	Defense Advanced Research Projects Agency
DBT	display builder tool
DE	domain engineering
DT&E	developmental test and evaluation
IDA	Institute for Defense Analyses
ITW/AA	Integrated Tactical Warning and Attack Assessment
OT&E	operational test and evaluation
QBT	query builder tool
RHMI	reusable human-machine interface
RICC	reusable integrated command center
RSP	Reuse-driven Software Process
SEE	software engineering environment
SEI	Software Engineering Institute
SPC	Software Productivity Consortium
STARS	Software Technology for Adaptable, Reliable Systems
UNAS	universal network architecture services

UNCLASSIFIED

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1996	Final Report, Feb 1995–Jun 1996

4. TITLE AND SUBTITLE

The Costs and Benefits of Domain-Oriented Software Reuse: Evidence From the STARS Demonstation Projects

5. FUNDING NUMBERS

DASW01 94 C 0054

A-144

6. AUTHOR(S)

Thomas P. Frazier and John W. Bailey

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER

IDA Paper P-3191

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12B. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

For more than twelve years, the Defense Advanced Research Projects Agency (DARPA) has funded software technology initiatives under its Software Technology for Adaptable, Reliable Systems (STARS) program. Between 1992 and 1995, DARPA funded three case studies to demonstrate the use of STARS technologies on three moderately sized software development projects, one in each military department. This paper describes the costs and benefits of using STARS technologies for these demonstration projects. IDA used goal-directed identification of measures and computed indicators of project status to ensure that relevant data and analyses were planned for each project. A major challenge was the difficulty with comparing the current projects to entirely analogous previous work by the same organization. Our results suggest the benefits of domain-oriented investments can be realized in three years. Potential savings of 75% to 90% in development costs could be attained after the initial investments have been recouped.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Technology for Adaptable, Reliable Systems (STARS); Software Engineering; Demonstrations; Cost Estimates	30
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR